Enhancing Workflow Efficiency Through DAG Merging and Parallel Execution in Apache Airflow

Anvith S G, Nithish Kushal Reddy, Sreebha Bhaskaran and Gurupriya M

Dept. of Computer Science and Engineering, Amrita School of Computing, Amrita Vishwa Vidyapeetham, Bengaluru, India

Keywords: Apache Airflow, Floyd-Warshall, DAGs, Union-Find

Abstract: Modern workflows are very complex, containing interdependencies and shared tasks that cause inefficiencies in the execution of tasks, utilization of resources, and dependency management. The structure of Directed Acyclic Graphs(DAGs) is very robust for modeling workflows but managing overlapping tasks across multiple DAGs results in redundancy, inconsistent dependencies, and long execution times. This work helps address the above issues with the development of a systematic way to merge multiple task-based DAGs into a single workflow that optimizes workflow execution. It verifies weak connectivity, cyclic properties, and consistency in dependencies between DAGs with algorithms such as Depth First Search(DFS)/Breadth First Search(BFS), Floyd-Warshall, Union-Find, In-Degree Similarity Check and Signature Hashing. Using the the merged DAG, Apache Airflow optimizes it for parallel execution, reducing the time taken to execute and maximizing resource usage. Each algorithm is assessed based on its time complexity, space complexity, and practical performance in order to determine the best solution for each stage. The final solution proves robust and scalable by deploying the integrated workflow in Apache Airflow, improving efficiency, removing redundancy, and optimizing the execution of tasks.

1 INTRODUCTION

Workflows become the backbones of modern data processing and task management systems, wherein efficient orchestration of tasks directly influences performance and resource utilization. DAGs are crucial in modeling workflows as nodes, which represent tasks, and edges, for which the dependencies between them are modeled. Shared tasks among multiple independent DAGs often make the workflow complex. If the workflows overlap with each other, how to actually handle those workflows becomes important to avoid redundancy for consistency and optimal resource usage-like energy, memory, or processing time. High-performance algorithms are used to solve the most important computational challenges in the work, including checking weak connectivity, acyclicity, and consistency of dependencies in the DAGs. The algorithms are then bench marked based on metrics like time complexity and space complexity to select the best methods in each step of the workflow merging procedure.

Workflows in the majority of actual usage are not separated silos, but interconnected systems, sharing common procedures. For example, pipelines in data engineering may contain common preprocessing steps shared across multiple workflows, or in bioinformatics, there may be several analyses that depend on the same preprocessing of the initial data. When such workflows are modeled as DAGs, common tasks become a source of redundancy, inefficient use of resources, and longer execution times. The motivation for this work comes from the need to improve the management of workflows by integrating various task-based DAGs into one optimized workflow. Such consolidation would remove redundancy, ensure that dependencies are handled consistently between shared tasks, optimize energy, memory, and time requirements.

Though modern workflows are commonly modeled as DAGs, representing workflows efficiently in terms of tasks along with their dependencies, challenges appear when multiple independent workflows share overlapping tasks or files, involving inefficiencies due to redundant task executions, inconsistencies in dependency resolution, and high

S G, A., Kushal Reddy, N., Bhaskaran, S. and M, G.
Enhancing Workflow Efficiency Through DAG Merging and Parallel Execution in Apache Airflow.
DOI: 10.5220/0013591500004664
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 3rd International Conference on Futuristic Technology (INCOFT 2025) - Volume 2*, pages 309-319
ISBN: 978-989-758-763-4
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

usage of resources. This must be overcome using a systematic approach to collapse multiple DAGs into a single, optimized workflow retaining the acyclic structure essential for dependency resolution. Given a set of DAGs representing individual workflows, where nodes are tasks and edges are dependencies, the problem is to merge those DAGs into one. The name of the nodes in different DAGs must be consistent with consistent dependencies; the merged DAG should avoid redundancy by removing duplicate computations, optimize resource usage in terms of energy, memory, and processing time, and support efficient parallelism in Apache Airflow. To address this problem, the work uses a multistage approach. First, it runs a Weak Connectivity Check to ensure that, when ignoring edge direction, each input DAG is structurally connected. Then, Acyclicity Verification verifies that all input graphs are valid DAGs. Finally, Dependency Consistency Check verifies that there is a class of consistent dependencies maintained by every set of common nodes over various DAGs. Once verified, DAG Merging will merge all the nodes and dependencies together in one coherent structure.
This work contributes to the following:

- An efficient method for merging DAGs with overlapping tasks is presented, such that consistency, acyclic properties, and redundancy elimination are ensured.
- Algorithms for merging are benchmarked for time and space complexities.
- The merged DAGs are implemented in Apache Airflow, which executes tasks in parallel.
- It shows optimization in the execution time and resource usage in general domains like healthcare and finance.

The work is structured as follows. Section II comprises the review of literature on the topic of DAG scheduling and optimization of workflow. Section III includes methodology for DAG merging and validation. Section IV comprises algorithmic framework of the approach along with time complexity analysis. Section V presents the results and benchmarking on the proposed algorithms. Finally, Section VI concludes the work undertaken along with the future research plan.

2 LITERATURE SURVEY

Efficient workflow management finds its application in many domains ranging from data analytics to scientific computing to cloud computing. DAGs are a basic graph model for representing complex workflows such that nodes represent tasks, and edges represent dependencies among them. Merging various DAGs into a workflow can be beneficial in terms of avoiding redundancy, ensuring constant dependency management, and exploiting resources. This literature review covers the key research efforts on DAG merging, workflow optimization, and task scheduling.

Shi and Lu (Shi and Lu, 2023) proposed performance models for large-scale data analytics DAG workflows, which can be executed in parallel across data. Their work deals with the resource allocation variability during execution and develops a Bottleneck Oriented Estimation (BOE) model to predict task execution time accurately. The model will help optimize the workflow's performance by detecting system bottlenecks. Sukhoroslov and Gorokhovskii (Sukhoroslov and Gorokhovskii, 2023) benchmarked DAG scheduling algorithms on scientific workflow instances. They have used a set of 150 real-world workflow instances for the evaluation of 16 scheduling algorithms with respect to performance in different cases. Their results help one to select proper scheduling strategy for complex workflows. Hua et al. (Hua et al., 2021) proposed a reinforcement learning-based approach to scheduling DAG tasks. Their algorithm iteratively adds directed edges to the DAG to enforce execution priorities, simplifying the problem of scheduling and improving performance. This method shows promise for the application of machine learning in workflow optimization. Zhang et al. have proposed a multi-objective optimization algorithm for DAG task scheduling in edge computing environments. This work takes into account parameters like the delay in completing tasks and energy consumption. Thus, the approach presents a balanced solution for constrained resources. It also explains the significance of multi-objective optimization in workflow management.

Kulagina et al. (Kulagina et al., 2024) tackled the challenge of executing large, memory-intensive workflows on heterogeneous platforms. The authors presented a four-step heuristic for partitioning and mapping DAGs with an objective to optimize makespan while respecting memory constraints. Their work improves the scalability and efficiency of workflow execution. Zhao et al. (Zhao et al., 2021) improved on DAG-aware scheduling algorithms by bringing efficient caching mechanisms onto the table, thus working through such a policy referred to as LSF, an improvement upon scheduling times for completed jobs to incorporate strategies optimized in cache management. Li et al. explored DAG-based task scheduling optimization in heterogeneous computing environments. They developed an improved algorithm based on the firefly algorithm, achieving better load balancing and resource utilization. This study underscores the need for tailored scheduling strategies in diverse computational settings.

Lin et al. (Lin et al., 2022) proposed AGORA, a scheduler that optimizes data pipelines in heterogeneous cloud environments. AGORA considers task-level resource allocation and execution holistically, achieving significant performance improvements and cost reductions. This work points out the benefits of global optimization in cloud-based workflows. Wang et al. studied DAG task scheduling using an ant colony optimization approach. Their model improves task migration and load balancing, enhancing the efficiency of heterogeneous multi-core processors. This research illustrates the application of bio-inspired algorithms in workflow scheduling. Zhou et al. presented a method of deep reinforcement learning for scheduling real-time DAG tasks. Their approach evolves scheduling policies that adapt the dynamic conditions of the systems, improving schedulability and performance. This shows the potential of deep learning in real-time workflow management. Kumar et al. proposed a deterministic model for predicting the execution time of Spark applications presented as DAGs. Their model enables resource provisioning and performance tuning, ensuring efficient workflow execution in big data platforms.

Gorlatch et al. proposed formalism to describe DAG-based jobs in parallel environments using process algebra. Their work represents a theoretical foundation for modeling complex workflows, enabling optimization efforts. Kumar et al. studied parallel scheduling strategies for data-intensive tasks represented as DAGs. Their algorithms consider locality of data and task dependencies in order to improve the execution efficiency within distributed systems. Zhang et al. proposed a multi-objective optimization algorithm for DAG-based task scheduling within edge computing environments. They balance between the delay for completing the tasks and the energy consumption within the system to provide an all-inclusive solution for such resource-constrained settings. Kulagina et al. (Kulagina et al., 2024) tackled the problem of executing big, memory-intensive workflows on heterogeneous platforms. They proposed a four-step heuristic for DAG partitioning and mapping: optimizing for makespan and respecting memory constraints. The approach improves the scalability and efficiency of workflow execution. The investigation of Hyperledger Fabric goes along the lines of focus of the uploaded paper: dependency consistency and redundancy elimination in DAGs. The Fablo-based implementation and IOTA Tangle's DAG technology is reflecting the use of DFS and BFS for real-time dependency and acyclicity checks.Optimized scheduling of tasks in cloud-based environments mirrors the paper on parallelism in Apache Airflow, which also improves on resource utilization. The Bayesian dual-route model is thus consistent with DAG merging for eliminating redundancy and ensuring that results are consistent, allowing for systematic benchmarking of improvements. Studies on DCN (dorsal cochlear nucleus) parallelize DAG optimization in Apache Airflow as both ensure resource efficiency and accuracy through structured frameworks—DCN for detecting features and DAGs for the execution of workflows.

The structure and dependency in Directed Acyclic Graphs (DAGs) can be used for the graph-based approach for effective anomaly identification and predicting the spread of attacks based on modeling workflows for resource optimization and strategies of execution. Optimized algorithms assure proper containment with respect to dependency and redundancy, providing scalability across diverse applications. Centrality-based adversarial perturbations exploit crucial nodes or edges to undermine graph neural networks (GCNs) with significant impact on the node classification. Countermeasures such as robust algorithmic designs and dependency validation checks enhance the resilience of the system without compromising efficacy in real-world deployments to perform the task orchestration efficiently.

Despite the progress in DAG scheduling, optimization, and orchestration, there are still many research gaps. Most of the existing works focus on specific aspects, such as scheduling efficiency or resource utilization, without integrating these with practical deployment. Approaches like reinforcement learning (Hua et al., 2021) and ant colony optimization are good at theoretical optimization but not scalable and adaptable in real time for dynamic environments. Moreover, the problem of DAG merging with overlapping tasks is not well explored, which leads to inefficiencies and waste of resources. Scalability solutions for memory-intensive workflows in distributed environments are also not plentiful. Though works such as (Kulagina et al., 2024) and have solutions to heterogeneous platforms, they depend heavily on particular architectural assumptions that diminish their adaptability. Moreover, optimization techniques in workflow within the framework of Apache Airflow are also underutilized, and very few studies bridge theoretical advancements in DAG optimization with practical implementation. This work introduces a novel, integrated approach to DAG merging and optimization within Apache Airflow.

It contrasts with most of the previous work in this area, which deals with isolated tasks or resource allocation. This one integrates multiple DAGs into a single optimized workflow. It removes shared task dependencies, eliminates redundancy, and allows for parallel execution. The algorithms on weak connectivity, acyclicity, and dependency consistency benchmark to ensure the most efficient approaches are being used and tailored for deployment in practice within Airflow. This work depicts distinctive, scalable task orchestration in Apache Airflow through the direct applicability of merged DAGs into the real-world environment. It systematically compares the algorithmic complexities to augment the depth of rigor and reproducibility, hence yielding actionable information to optimize workflows for future use.

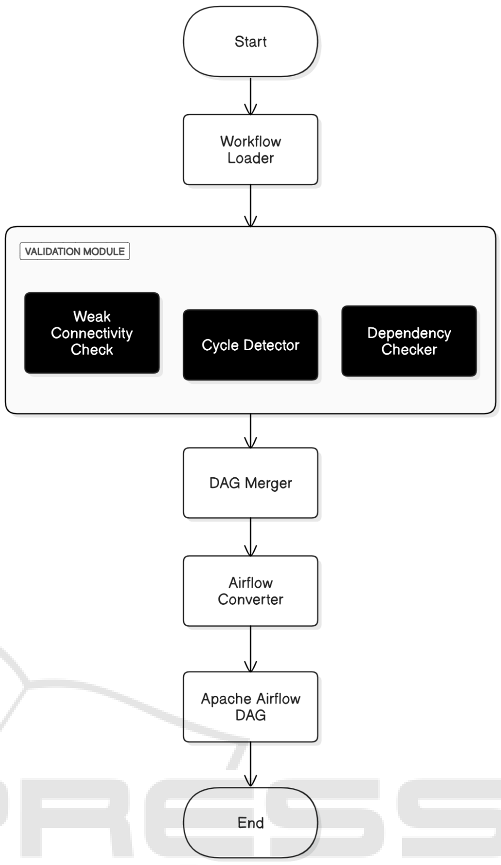


Figure 1: System Architecture

3 METHODOLOGY

The methodology of the work, Enhancing Workflow Efficiency through DAG Merging and Parallel Execution in Apache Airflow, is to design a structured approach that will merge multiple DAGs into an optimized single workflow. It helps in ensuring efficient execution and eliminates redundancy and supports parallel processing in Apache Airflow. The methodology involves several stages, each aimed at addressing specific aspects of the merging and optimization process.

3.1 System Overview

The system optimizes workflow execution by merging multiple DAGs into a single, unified workflow for efficient parallel execution in Apache Airflow. It begins with loading and preprocessing the input workflows, represented as DAGs, into a standard format as shown in Fig. 1. Each of the DAGs is subject to weak connectivity checks in order to ensure structural cohesion, as well as acyclicity verification to ensure that there are no cycles. After such validation, which is guaranteed by in-degree mapping and adjacency matrix comparisons, merged DAGs, maintaining properties of acyclicity with removal of redundancy, become a unified structure. Merging all the above procedures, one now converts the obtained merged DAG into the appropriate format required to be Apache Airflow compatible. Here nodes become tasks, and edges represent their dependencies. The workflow will be deployed in Airflow to orchestrate parallel tasks and have performance analysis for execution time, resource utilization, and reduction of redundancy, leading to scalability, efficiency, and smooth execution of complex workflows.

3.2 Preprocessing and Input Representation

To start with, the input workflows are represented as DAGs. Each DAG is defined by its tasks, which are nodes, and their dependencies, which are edges. The workflows are then converted into a standard format by using adjacency lists, where each task, or node, is mapped to its immediate dependencies. A directed graph representation, such as networkx. DiGraph, is used to ensure consistency and ease of manipulation throughout the methodology. This is also where the merge assumptions are made. The nodes with the same name across distinct DAGs are assumed to be the same task, and they have consistent sets of dependencies. Any inconsistency in the assumption is taken care of during the dependency consistency check.

3.3 Weak Connectivity Check

Verifies that each DAG has weak connectivity, meaning that in a single DAG, all the tasks are structurally connected if the directions of the edges are ignored. Thus, this check ensures that there are no isolated components in the individual workflows. This is accomplished by reducing each digraph into a nondig and executing DFS or BFS in order to determine whether the nodes can be reached from any of them. If any such DAG is strongly disconnected, the merging stops, and adjustments must be made so that the original workflows will have the structures corrected.

3.4 Acyclicity Verification

To verify the validity of the input workflows, each DAG should be checked for the nonexistence of cycles. A valid DAG should not have any directed cycles because they show cyclic dependencies that cannot be solved. It follows two approaches: DFS by recursion stack and Floyd-Warshall algorithm. The method for the DFS approach checks a back edge during traversal, that means a cycle, and Floyd-Warshall makes a reachability matrix where it checks for diagonal entry that means cycle. For this purpose, these two approaches are compared in time and space complexity for large workflow for the selection of appropriate method.

3.5 Dependency Consistency Check

The next step ensures that the DAGs agree on shared work items by comparing dependencies. For every node in two or more DAGs, the set of dependencies of each node is extracted and then compared across all DAGs. All of these conflicts must be resolved before the merge process continues. This consistency check employs in-degree maps, whereby for each node, the mapping to its predecessor nodes is employed as a representation of the set of dependencies. The in-degree maps were hashed using efficient hashing algorithms to allow for fast comparisons. This step was fundamental to maintaining the correctness of the merged DAG and avoiding faults at execution time.

3.6 DAG Merging

Once the input DAGs are validated for having consistent dependencies, merging can begin. The resultant workflow of all the input DAGs is merged so that nodes and their dependencies appear together in a unified, acyclic graph. In iterative processing of each inputting DAG, the algorithm addes its nodes and edges within it to a global dependency. Shared nodes are taken into accounts by merging their dependencies into respective results without duplication. Network is then used to represent these dependencies in a direct graph form. This resultant merging Dag represents the whole workflow together in a coherent structure of execution.

3.7 Conversion to Apache Airflow DAG

Translated to an Apache Airflow-compatible format, such that the DAG's final merged and optimized form. Each node of the graph is represented by an Airflow task, while the edges correspond to the task dependencies between the nodes of the graph. The Airflow DAG is a Python module, with proper observance to the Airflow's API. The automated script, that will generate the Airflow DAG file, has been provided for integration with the system. This ensures that the workflow is directly deployable and executable in the task orchestration environment of Airflow.

3.8 Algorithm Comparison and Benchmarking

The methodology makes use of several algorithms for every task, comparing their time and space complexities to identify the best approaches to fulfill the needs of the work. Algorithms such as DFS/BFS and Union-Find for the Weak Connectivity Check will be compared in their trade-offs regarding computational and resource efficiency. Acyclicity Verification can be benchmarked against a DFS using a recursion stack for Floyd-Warshall for detecting cycles. For Dependency Consistency Check, various techniques such as signature hashing and direct comparison are investigated in order to determine which of them is the best technique to check consistency in the nodes. Benchmarking results provide an in-depth view of computational efficiency and resource utilization, guiding the choice of optimal algorithms for each task.

4 WORKFLOW INTEGRATION AND OPTIMIZATION FRAMEWORK

In this work, it uses the following complete algorithm toolkit for each step of the DAG merging and execution process. Each algorithm has been chosen to complete a given computational task with efficiency but guaranteed correctness. The algorithms are outlined as follows in detail with functionality and methodology :

4.1 Weak Connectivity Check

The following provides three algorithms for testing if a graph is weakly connected, namely DFS/BFS and Floyd-Warshall with Union-Find that detect whether each of the graph is a connected directed graphs when their direction arrows are ignored. The DFS/BFS Algorithm first converts the given directed graph into an undirected graph. That is, it simply ignores the direction of edges. From any given node, it performs a traversal that marks all reachable nodes. If the count of visited nodes is equal to the total count of nodes in the graph, then the DAG is weakly connected. This technique uses recursive or iterative traversal to explore the graph appropriately. The Floyd-Warshall Algorithm constructs an adjacency matrix for the undirected version of the graph and iteratively updates the matrix to figure out all-pair reachability. If no pair of nodes is reachable to one another, then graph identifies as weakly disconnected. More time and memory resources is utilized by this algorithm than required in DFS/BFS. The Union-Find algorithm uses the disjoint-set data structure to perform group node operations into connected components. Each node is its own parent initially and process edges to union nodes into the same component. At the end, when all nodes share the same root, the graph is weakly connected. This method is extremely efficient for large graphs with many edges.

4.2 Acyclicity Verification

To check that the input graphs are valid DAGs, there are two algorithms that are implemented: DFS with Recursion Stack and Floyd-Warshall. Both assume the strategy of detecting cycles because a DAG cannot have cycles. The DFS with Recursion Stack Algorithm uses a depth-first search along the graph, keeping a recursion stack of nodes currently visited. If a back edge shows up (i.e., a node within the recursion stack is visited again), then there must be a cycle, so the graph isn't a DAG. The algorithm follows the entire graph in order that the nodes are all cycle-free, and thus it is at the same time correct and efficient. Floyd-Warshall algorithm is another algorithm that uses the reachability matrix to identify the cycles. The matrix starts recording all the direct edges between nodes and then iteratively updates itself to include indirect paths through other nodes. A cycle exists if a diagonal entry of the matrix equals 1 because a node can reach itself. It is more computationally intensive but has a mathematical guarantee of being acyclic.

4.3 Dependency Consistency Check

To ensure that shared nodes across multiple DAGs have consistent dependencies, three algorithms are used: In-Degree Similarity Check, Signature Hashing, and Adjacency Matrix Comparison. Each algorithm addresses the task of dependency consistency verification through different methodologies, ensuring robustness and accuracy across diverse scenarios. The In-Degree Similarity Check computes the in-degree map for each graph, mapping every node to its set of predecessor nodes (dependencies). This map represents the task dependencies in a structured format. For shared nodes appearing in multiple DAGs, the algorithm compares their in-degree sets across all graphs.

If any discrepancies are found—indicating that a shared node has different dependencies in different DAGs—the algorithm flags an inconsistency. This approach systematically evaluates each shared node, ensuring correctness by exhaustively checking all in-degree sets. The computational process involves iteration over all nodes in each graph, constructing the in-degree maps, and comparison. Although this method ensures accuracy, it can become computationally expensive for large graphs with many shared nodes. Using this concept of in-degree maps, the Signature Hashing Algorithm improves efficiency with the introduction of hash-based comparisons. In this algorithm, the in-degree sets for every node are sorted first, and then hashed by the help of a cryptographic hash function, such as MD5. This results in hash signatures acting as unique identifiers for the dependence set of each node.

For shared nodes, the algorithm compares hash signatures of each node across all DAGs. The presence of more than one hash signature of a node signifies inconsistent dependencies. It reduces the size of in-degree sets to a concise hash value that reduces the overall computational overhead for comparisons and hence is ideal for large graphs with complex dependencies. The Adjacency Matrix Comparison Algorithm represents a global view of dependency consistency by using adjacency matrices.

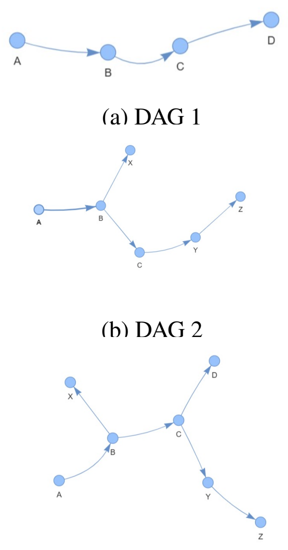
The algorithm first generates an adjacency matrix for every DAG, which presents a tabular view of the graph structure, where rows and columns are the nodes, and cell values indicate the existence of edges. This algorithm compares column vectors corresponding to shared nodes in all adjacency matrices to ensure consistency. If the columns do not match then inconsistent dependencies for the shared node exist. The approach is computationally expensive due to a matrix generation and comparison process but gives a complete view of the graph structure. Adjacency matrix approach is very effective for small to medium-sized graphs where the computational cost is easy to handle, and the whole structural comparison is envisioned.

4.4 DAG Merging

Used for the aggregation of multiple DAGs into one combined workflow. The Dependency Aggregation Algorithm iterates all input DAGs, which then aggregate their nodes and edges into a global dependency list. Nodes with identical names are merged and dependencies are aggregated into a set. The resulting dependency list is translated to a directed graph using the networkx. The DiGraph representation of the merged workflow to preserve its acyclic structure is shown in Figs. 2 (a), (b) and (c). Fig 2 (a) and (b) are the DAG's which are yet to merge. Fig 2 (c) is the final merged DAG. The algorithm is designed to address the conflicts that may result from merging and consistencies checks on the shared nodes. It ensures that the resulting DAG is valid and redundant computations or tasks exist.

4.5 Conversion to Apache Airflow DAG

The merged DAG is translated into an Airflow format for enabling execution. It maps every node of the DAG into an Airflow task and translates all the dependencies into the Airflow DAG structure. The output will be Python code utilizing the task and DAG APIs, so the deployment and execution will go very smoothly. This algorithm recognizes the independent tasks and optimizes for parallel execution through the grouping of tasks with no dependencies. In this manner, the workflow leverages the parallelism capability of Airflow to result in minimal execution time with proper resource utilization.



(a) DAG 1

(b) DAG 2

(c) Merged DAG

Figure 2: Illustration of DAG merging process: (a) Original DAG 1, (b) Original DAG 2, (c) Merged DAG

Each algorithm used within this work is designed to resolve specific computational challenges associated with DAG merging and execution. From checking weak connectivity and acyclicity to ensuring dependency consistency and merging workflows, each of these algorithms provides the bedrock for the methodology that will be followed. Systematic application ensures correctness, efficiency, and scalability in a unified workflow, which easily integrates and executes in Apache Airflow.

5 RESULTS AND ANALYSIS

Analysis of algorithms for checking weak connectivity, acyclicity verification and dependency consistency check considers both theoretical dimensions: the time complexity and space complexity, as well as empirical timing experiments on representative test inputs. Investigating both aspects together reveals how each algorithm scales and behaves in real-world scenarios, in contrast to what complexity analysis predicts.

5.1 Weak Connectivity Check

Testing on different graph sizes (10 nodes/20 edges, 15 nodes/40 edges, 20 nodes/50 edges, and 50 nodes/200 edges) demonstrated that DFS/BFS and Union-Find ran consistently in the range of microseconds to a few milliseconds. Even on 50 nodes and

200 edges, DFS/BFS took about 0.001034 seconds and Union-Find about 0.001105 seconds. Floyd-Warshall, though correct, took around 0.023842 seconds on the largest graph tested, showing its relative inefficiency at scale. From a theoretical point of view, DFS/BFS is O(n + m), where n is the number of nodes and m is the number of edges. This fits well with the observed near-linear growth in runtime. Union-Find's almost linear complexity $O(m \cdot \alpha(n))$ (with $\alpha(n)$ being very slow-growing) similarly matches the practical results, showing minimal runtime growth. Contrastingly, Floyd-Warshall's $O(n^3)$ complexity manifested itself in a tremendous increase in execution time with increasing problem size – practical evidence that this method doesn't scale well for large workflows. Both theory and practice affirm that DFS/BFS and Union-Find are best suited for large workflows and maintain low execution times. Floyd-Warshall, though theoretically perfect for dense, small graphs, is impractical when the graph grows.

5.2 Acyclicity Verification

In the test cases presented, the DFS recursion stack method correctly identified cycles nearly immediately, typically finishing in less than 0.0001 seconds. Floyd-Warshall correctly found cycles as well, but even for relatively small graphs (50 nodes/200 edges), it took about 0.010919 seconds—a very fast time, but clearly slower than DFS. Theoretically, DFS with recursion stack runs in O(n + m), which is good for sparse to moderately dense workflows. Floyd-Warshall's $O(n^3)$ complexity is quite significant. For small graphs, the difference in time might be insignificant, but as graphs grow, the gap widens, which matches the experimental observations. The alignment between theory and practice is clear. Floyd-Warshall guarantees a global perspective on reachability, but its cubic complexity leads to slower practical runtimes as graphs grow. The DFS recursion stack approach, in both theory and empirical testing, is the better choice for verifying acyclicity in larger workflows.

5.3 Dependency Consistency Check

In case of multiple lists of dependency, the In-Degree Similarity Check has been shown to be the fastest method, consuming approximately 0.000182 seconds for 5 lists and 0.000650 seconds for 20 lists. Signature Hashing consumed more time, having an overhead from the hashing operations-approximately 0.000794 seconds (5 lists) and 0.002018 seconds (20 lists). Adjacency Matrix Comparison was the slowest (0.005147 seconds for 5 lists and 0.009037 seconds for 20 lists) and showed to be much more computationally and memory-intensive.

The In-Degree Similarity Check runs at $O(k \cdot (n+m))$, which is quite good for sparse graphs and scales pretty well in practice. Signature Hashing at $O(k \cdot n \cdot \log(n))$ does introduce hashing overhead but is still quite efficient. Adjacency Matrix Comparison, $O(k \cdot n^2)$, becomes cumbersome for larger graphs as the matrix-based approach matches the observed slowdown in practical tests. Predictions on the theoretical level closely relate with experiments performed. In-Degree Similarity Check and Signature Hashing efficiently scale and stay practical while the number of DAGs and their sizes increase. Adjacency Matrix Comparison presents quadratic complexity and high memory requirements, which becomes a bottlenecks in practical timings.

5.4 DAG Merging

DAG Merging, utilizing a dependency aggregation approach, comes with a time complexity of O(n + m) and a space complexity of O(n + m). Although the test results did not give direct timing results recently, the complexity analysis indicates that it should scale linearly. Since the other stages match their theoretical and practical results, it is also expected that this step will be efficient and have no overhead. At each step, empirical results have corroborated theoretical complexity analysis. Simpler, near-linear algorithms such as DFS/BFS, DFS stack cycle detection, and In-Degree Similarity Check proved to be theoretically guaranteed, being efficient in practice. More complex algorithms such as Floyd-Warshall and Adjacency Matrix Comparison, which were theoretically sound and complete, ran relatively slow in real-time testing, so the observed runtime behavior was almost entirely explained by theoretical complexity.

Finally as shown in Table 1, Table 2 and Table 3 this alignment between theoretical complexities and practical measurements can provide very useful guidance for choosing the appropriate algorithm. For large complex workflows, algorithms with lower theoretical complexity consistently deliver faster real-world performance, so it is ensured that the workflow solution integrated and deployed in Apache Airflow is efficient and scalable.

Table 1: Algorithm comparison for dag operations.

Task	**Algorithm**		**Space Complexity**
Weak Connectivity Check	DFS/BFS	O(n+m)	O(n+m)
	Floyd-Warshall	$O(n^3)$	$O(n^2)$
	Union-Find	O(m·(n))	O(n)
Acyclicity Verification	DFS with Recursion Stack	O(n+m)	O(n)
	Floyd-Warshall	$O(n^3)$	$O(n^2)$
Dependency Consistency Check	In-Degree Similarity Check	O(k·(n+m))	O(k·n)
	Signature Hashing	O(k·nlog(n))	O(k·n)
	Adjacency Matrix Comparison	$O(k \cdot n^2)$	$O(k \cdot n^2)$
DAG Merging	Dependency Aggregation	O(n+m)	O(n+m)

Table 2: Execution time comparison of dependency consistency check methods.

Method	**5 Lists**	**20 Lists**
In-Degree Similarity Check	0.000182 s	0.000650 s
Adjacency Matrix Comparison	0.005147 s	0.009037 s
Signature Hashing	0.000794 s	0.002018 s

5.5 DAG Merging in Apache Airflow

For testing the proposed methodology as feasible and efficient, an example workflow orchestration with Apache Airflow was performed. This synthetic sales dataset example consists of features such as product_id, product_name, quantity, price, category, rating, and best_sales_channel that allow one to calculate revenues, analyze the ratings of the customers, and predict the channel through which most sales occur. Four workflows were designed: three independent workflows (DAG 1, DAG 2, DAG 3) and a combined, optimized workflow (DAG 4), as shown in Figure 3, Figure 4, Figure 5, and Figure 6, respectively.

The independent workflows have duplicated tasks. For instance, both DAG 1 and DAG 2 share some common initial tasks (load_data and transform_data) but take different paths as downstream tasks in that DAG 1 performs revenue related calculations (calculate_product_revenue and calculate_total_revenue) while DAG 2 is doing rating model training for ratings using the following train_ml_model_on_rating, predict_and_save_rating, and test_model_accuracy_on_rating. Likewise, channel specific machine learning tasks are carried out by DAG 3 using train_ml_model_on_channel, predict_and_save_channel, and test_model_accuracy_on_channel. This redundancy in shared tasks leads to inefficiencies in resource usage and execution time when workflows are executed independently.

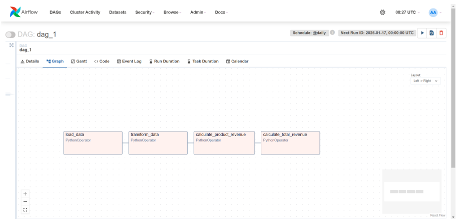


Figure 3: DAG 1

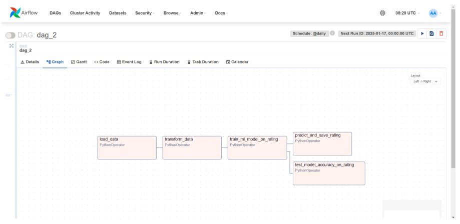


Figure 4: DAG 2

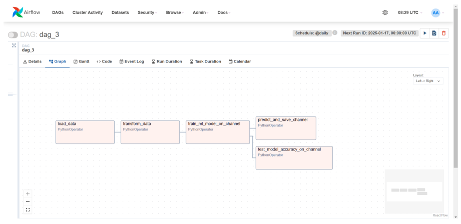
The merged workflow, DAG 4 (Figure 6), combines shared tasks into a single execution, removing redundancy and increasing efficiency. Common preprocessing tasks like load_data and transform_data are performed only once, and all downstream domain-specific tasks are built upon this. Then, there are separate branches for revenue calculation, ratings analysis, and channel-based predictions that run independently and in parallel. The DAG merged ensures consistent resolution of dependencies and leverages the parallelism of Apache Airflow for better scalability and performance. This methodology, it is obvious, is sound for streamlining complex workflows in large-scale data processing.

6 CONCLUSION

Develop and implement a structured method to combine several DAGs with common tasks into one optimized workflow that can be run in parallel using Apache Airflow. The method addresses some of the most critical computational problems: checking weak connectivity, ensuring acyclic structures, and preserving consistent dependencies on shared tasks. Advanced algorithms are developed and benchmarked, and the work discovers optimal solutions with respect to the structural properties of the input DAGs. The unified workflow resulting from this merger eliminates redundancy, ensures accuracy, and optimizes the use of resources in terms of energy, memory, and execution time. This is well demonstrated in the deployment using Apache Airflow, where the merged DAG is proven to be practical in the management of complex workflows in the

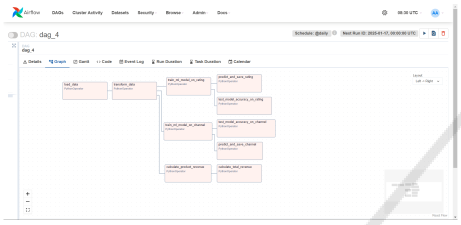
Table 3: Performance comparison of graph algorithms.

Graph	**DFS (s)**	**Floyd-Warshall (s)**	**Union-Find (s)**	**DFS Cycle (s)**	**Floyd Cycle (s)**
10 nodes, 20 edges	0.000299	0.004817	0.000315	0.000024	0.000229
15 nodes, 40 edges	0.000248	0.001250	0.000263	0.000011	0.000360
20 nodes, 50 edges	0.000289	0.002329	0.000328	0.000013	0.000830
50 nodes, 200 edges	0.001034	0.023842	0.001105	0.000013	0.010919



Schedule: @daily
Next Run ID: 2025-01-17, 00:00:00 UTC

Figure 5: DAG 3


Airflow
DAGs
Cluster Activity
Datasets
Security
Browse
Admin
Docs
08:30 UTC
AA
DAG: dag_4
Details
Graph
Gantt
Code
Event Log
Run Duration
Task Duration
Calendar
Layout: Left -> Right
load_data PythonOperator
transform_data PythonOperator
train_ml_model_on_rating PythonOperator
predict_and_save_rating PythonOperator
test_model_accuracy_on_rating PythonOperator
train_ml_model_on_channel PythonOperator
test_model_accuracy_on_channel PythonOperator
predict_and_save_channel PythonOperator
calculate_product_revenue PythonOperator
calculate_total_revenue PythonOperator

Schedule: @daily
Next Run ID: 2025-01-17, 00:00:00 UTC

Figure 6: DAG 4

finance, healthcare, and data engineering domains.

This work forms a basis for considerable workflow optimization and task orchestration progress, opening up to dynamic and adaptive methodologies. Future research directions would involve developing a dynamic adaptation feature within workflows where the merged DAG would be updated in real time depending on the changing dependencies, the evolution of priorities among tasks, or new requirements placed on workflows. This would enhance flexibility and responsiveness of workflow management systems in the optimum allocation of resources and its subsequent execution in changing environments. Even, machine learning techniques may come into use; for instance, reinforcement learning that can predict a change in conditions of the workflow to avoid more and more interruptions with the purpose of maximum efficiency. This would further open up the space of applicability by integrating with other workflow management systems, for example, Prefect or Luigi; extension into domains such as IoT and health care would open up the opportunity to dynamically coordinate tasks in sensor networks or pipelines of patient data.

REFERENCES

Shi, J., Lu, J.: Performance models of data parallel DAG workflows for large scale data analytics. *Distrib. Parallel Databases* 41(3), 299–329 (2023)

Sukhoroslov, O., Gorokhovskii, M.: Benchmarking DAG scheduling algorithms on scientific workflow instances. In: *Russian Supercomputing Days*, pp. 3–20. Springer, Cham (2023)

Hua, Z., et al.: Learning to schedule DAG tasks. *arXiv preprint arXiv:2103.03412* (2021)

Tong, Z., et al.: Multi–objective DAG task offloading in MEC environment based on federated DQN. *IEEE Trans. Serv. Comput.* (2024)

Kulagina, S., et al.: Mapping large memory-constrained workflows onto heterogeneous platforms. In: *Proc. ICPP*, pp. 1–10 (2024)

Zhao, Y., et al.: Performance improvement of DAG-aware task scheduling in Spark. *Electronics* 10(16), 1874 (2021)

Chen, C., Zhu, J.: DAG-based task scheduling optimization in heterogeneous systems. In: *CCF Conf. CSCW*, pp. 45–58. Springer, Singapore (2023)

Lin, E., et al.: Global optimization of data pipelines in cloud environments. *arXiv preprint arXiv:2202.05711* (2022)

Yi, N., et al.: Task scheduling using improved ant colony algorithm. *Future Gener. Comput. Syst.* 109, 134–148 (2020)

Sun, B., et al.: Edge generation scheduling using deep RL. *IEEE Trans. Comput.* (2024)

Tariq, H., Das, O.: Deterministic model for Spark execution prediction. In: *Eur. Workshop PE*, pp. 112–126. Springer (2022)

Barbierato, E., et al.: Map–reduce process algebra for DAG jobs. In: *ASMTA*, pp. 78–92. Springer (2020)

Meng, X., Golab, L.: Parallel scheduling of data-intensive tasks. In: *Euro-Par*, pp. 203–217. Springer (2020)

Deng, W., et al.: Multi–objective DAG scheduling in fog computing. *Wirel. Netw.* (2024)

Kulagina, S., et al.: Memory-intensive workflow execution. In: *ICPP*, pp. 1–10 (2024)

Pranesh, R., et al.: Securing IoT using Hyperledger Fabric. In: *ICCCNT*, pp. 1–7. IEEE (2024)

V, A., et al.: EdgeGuard: Spam detection for IOTA Tangle. In: *INCET*, pp. 1–9. IEEE (2024)

Choudhary, S., et al.: Enhanced task scheduling in logistics. In: *MysuruCon*, pp. 1–7. IEEE (2022)

Meng, X., Golab, L.: Parallel scheduling strategies. In: *Euro-Par*, pp. 203–217. Springer (2020)

Deng, W., et al.: Container migration in fog computing. *Wirel. Netw.* (2024)

Nair, A.B., et al.: Centrality attacks on GCNs. *J. Netw. Secur.* (2024)

Subhakkrith, S.R., et al.: Graph-based anomaly detection. In: *ICCCNT*, pp. 1–8. IEEE (2024)